

Diamond Dicing

Hazel Webb^a, Daniel Lemire^b, Owen Kaser^a

^a*University of New Brunswick Saint John*

^b*Université du Québec à Montréal*

Abstract

Many queries that constrain multiple dimensions simultaneously are difficult to express and compute efficiently in both Structured Query Language (SQL) and multi-dimensional languages. We introduce the diamond cube operator, filling a gap among existing data warehouse operations. It aids decision support and analysis by facilitating the execution of simultaneous multidimensional queries on data cubes. An example query from the business analysis domain follows.

A company wants to close shops and terminate product lines simultaneously. The CEO wants the maximal set of products and shops such that each shop would have sales greater than \$400 000, and each product would bring revenues of at least \$100 000—assuming we terminate all other shops and product lines.

Diamonds are intrinsically multidimensional and because of the interaction between dimensions the computation of diamond cubes is challenging. We present diamond dicing experiments on large data sets of more than 100 million facts. Our custom Java implementation is more than one hundred times faster, on a large data set, than a popular database engine.

Keywords: OLAP, information retrieval, multidimensional queries

1. Introduction

Dealing with large data is challenging. One way to meet this challenge is to choose a sub-sample—select a subset of the data—with desirable properties such as representativity, conciseness, or homogeneity. In signal and image processing, software sub-samples data [1] for visualisation, compression, or analysis purposes: commonly, images are cropped to focus the attention on a particular segment. In databases, researchers have proposed similar sub-sampling techniques [2, 3], including iceberg queries [4–6] and top- k queries [7, 8].

Such reduced representations are sometimes of critical importance to get good on-line performance in Business Intelligence (BI) applications [4]. Even when performance is not an issue, browsing and visualising the data frequently benefit from reduced views [9].

	Chicago	Montreal	Miami	Paris	Berlin	Totals
TV	3.4	0.9	0.1	0.9	2.0	7.3
Camcorder	0.1	1.4	3.1	2.3	2.1	9.0
Phone	0.2	6.4	2.1	3.5	0.1	12.3
Camera	0.4	2.7	5.3	4.6	3.5	16.5
Game console	3.2	0.3	0.3	2.1	1.5	7.4
DVD player	0.2	0.5	0.5	2.2	2.3	5.7
Totals	7.5	12.2	11.4	15.6	11.5	58.2

Figure 1: Sales (in million dollars) with a 5,10 sum-diamond shaded: stores need to have sales above \$10 million whereas product lines need sales above \$5 million.

The computation of icebergs, top- k elements, or heavy-hitters has received much attention [10–12]. This type of query can be generalised so that interactions between dimensions are allowed. We propose a model that has acceptable computational costs and a theoretical foundation. We illustrate the proposal with examples from BI and from bibliometrics.

BI example. In the BI context, consider Fig. 1, which represents the sales of different items in different locations. Typical iceberg queries might be requests for stores having sales of at least 10 million dollars or product lines with sales of at least 5 million dollars. However, what if the analyst wants to apply both thresholds simultaneously? He or she might contemplate closing both some stores and some product lines. In our example, applying the constraint on stores would close Chicago, whereas applying the constraint on product lines would not terminate any product line. However, once the shop in Chicago is closed, we see that the product line TV must be terminated which causes the closure of the Berlin store and the termination of two new product lines (Game console and DVD player).

This multidimensional pruning query selects a subset of attribute values from each dimension that are simultaneously important. For all the attribute values remaining, every column (slice) in the location dimension sums to at least 10 and every row (slice) in the product dimension sums to at least 5.

For ease of visualisation, the example above is provided in two dimensions, but the diamond cube operator is defined over d dimensions. For example, a three-dimensional query applied in the same BI context might constrain products, stores and manufacturers.

The proposed operation is a *diamond dice* [13, 14].¹ It produces a *diamond*, as formally defined in Section 3. Fig. 1 shows a “5,10 sum” diamond.

Bibliometrics example. As another motivating example, consider a bibliographic table with columns for author, venue, and (first) keyword. Perhaps we want to analyze the publication habits of professors, but much work would be required to precisely identify

¹This article is an expanded version of a conference paper [14].

Table 1: Characteristics of selected diamonds from DBLP

Diamond	authors	venues	keywords	% size	
				reduction	interpretation
5,1,1	115206	5078	36323	32	professors
5,50,1	112058	3069	34501	35	professors, non-student venues
5,50,10	105986	2960	7299	39	profs, n.-s. venues, mainstream
119,119,1	11	7	482	>99.9	a prolific h/w cluster
100,100,2	16	11	333	>99.9	closely related h/w cluster
108,20,6	2	8	16	>99.9	two surprising h/w authors
86,20,7	317	472	920	98.5	hw/db/parallel/mmedia cluster
43,43,43	1513	591	535	96	much CS

which authors are professors. However, perhaps we can assume that most authors without at least 5 publications are not professors. A “5,1,1 count” diamond will exclude them. Moreover, suppose we worry that there might be venues that mostly cater to students. Publications in such venues should be excluded from analysis. The 5,50,1 diamond will ensure that every retained person has published at least 5 articles in retained venues, and that every retained venue has at least 50 (presumed) professors publishing there. Not only will the 5,50,1 diamond prune some venues, but it may also remove some pseudo-professors who relied on student venues to achieve 5 publications. The 5,50,10 diamond will additionally exclude publications on “fringe topics”—a query for that diamond is an attempt to identify publications on mainstream topics by professors in non-student venues.

For illustration, we processed conference publication data available from DBLP [15]; the data and details of its preparation are given elsewhere [16]. See Table 1 for some characteristics of diamonds on this data.

We can obtain other information about clusters of authors, venues and topics by requiring very large thresholds. For instance, the 119,119,1 diamond contains authors with 119 publications each in retained venues, and retained venues each have 119 publications from retained authors. This diamond contains 11 prolific authors in the area of digital hardware and computer-aided design, who publish in 7 venues using 482 keywords. There are more authors (16) and venues (11) in the 100,100,2 diamond, but it is still very much a hardware area with 333 keywords.

An extreme case is the 108,20,6 diamond, which involves the publications of authors I. Pomeranz and S. M. Reddy in 8 hardware venues. Within this diamond, the most frequent keyword is ‘synchronous’, which occurred 7 times more often than the least frequent, ‘sequential’. Globally (i.e., in the 0,0,0 diamond), these keywords are almost equally frequent and are ranked 286th and 289th. These two authors are ranked 20th and 11th, globally.

Setting high thresholds is particularly useful in obtaining smaller, more easily analyzed, sets of data. This motivates the problem of finding a $k,k,1$ (or $k,k,2$) diamond where the system is responsible for discovering that $k = 119$ (respectively, $k = 100$) results in the non-empty diamond with the fewest attribute values.

2. Related Work

Diamonds are related to other work in various ways. They extract important sub-samples of the data as in Formal Concept Analysis or skyline and top- k queries. Diamonds generalise a two-dimensional iterative pruning algorithm [17] that trawls the Web for cyber-communities. We explore these relationships in the following sections.

2.1. Trawling the Web for Cyber-communities

A specialisation of the diamond cube is found in Kumar et al.’s work searching for emerging social networks on the Web [17]. Our approach is a generalisation of their two-dimensional ITERATIVE PRUNING algorithm. Diamonds are inherently multidimensional. Kumar et al. [17] model the Web as a directed graph and seek large dense bipartite subgraphs. A bipartite graph is dense if most of the vertices in the two disjoint sets, U and V , are connected. Kumar et al. hypothesise that the signature of an emerging Web community contains at least one “core”, which is a *complete* bipartite subgraph with at least i vertices from U and j vertices from V . In their model, the vertices in U and V are Web pages and the edges are links from U to V . Seeking an (i, j) core is equivalent to seeking a *perfect* two-dimensional diamond cube (all cells are allocated).

The iterative pruning algorithm is a specialisation of the basic algorithm we use to seek diamonds: it is restricted to two dimensions and is used as a preprocessing step to prune data that cannot be included in the (i, j) cores. Although their paper has been widely cited [18–23], it appears that our work is the first to propose a multidimensional extension to their approach and to provide a formal analysis.

2.2. Sub-sampling with Database Queries

Relational Database Management Systems (RDBMS) have optimisation routines that are especially tuned to address both basic and more complex SELECT ... FROM ... WHERE ... queries. However, there are some classes of queries that are difficult to express in SQL, or that execute slowly, because suitable algorithms are not available to the underlying query engine. They include skyline, top- k and nearest-neighbour queries.

2.2.1. Iceberg Queries

The iceberg query introduced by Fang et al. [4] performs an aggregate function over a specified dimension list and then eliminates aggregate values below some specified threshold. Diamond dicing eliminates *slices* where the aggregate value of the *slice* falls below a threshold. Unlike icebergs, dimensions can be constrained by different thresholds simultaneously.

2.2.2. Skyline Operator

The Skyline operator [22, 24] seeks a set of points where each point is not “dominated” by some others: a point is included in the skyline if it is as good or better in all dimensions and better in at least one dimension. Attributes, e.g. distance or cost, must be ordered. Therefore, indexes can be beneficial to Skyline queries.

2.2.3. *Top-k*

Another query, closely related to the skyline query, is that of finding the “top- k ” data points. For example, we may seek the ten most popular products sold in a store. Donjerkovic and Ramakrishnan [25] frame the top- k problem as one of finding the best answers quickly rather than all the answers. Their method is applicable to unsorted and unindexed attributes. (Otherwise the index would be used to provide an exact result.)

Top- k dominating queries [26] combine skyline and top- k to present a set of points that dominate others in all dimensions, but keep the result set to a user-defined size.

2.2.4. *Nearest Neighbours*

One of the most common multidimensional queries is the nearest neighbour query, which seeks elements that are “close” to a provided target. For example, given a set of users, we might seek users that have a profile similar to the current user. A common query asks to find the k nearest neighbours (kNN), that is, k neighbours that are as close as possible to the target.

Reverse nearest neighbours [27] start with a given element and asks which possible targets would have this element in the nearest neighbours. For example, imagine that customers only visit one of the 10 nearest stores. Given a customer, which store locations would attract him?

Nearest neighbour queries are often difficult to index, though some hashing techniques [28, 29] can help accelerate queries. They also require a specific distance measure.

2.2.5. *Outlier Identification*

Another frequent type of query in multidimensional data sets is outlier identification. For example, we might seek elements which are far from most other data points [30]. Points that are not outliers are “near” some fraction of the other data sets. That is, we are left with a “core”. Meanwhile a diamond cube can be seen as such a core.

2.3. *Formal Concept Analysis*

In Formal Concept Analysis [31, 32] a Galois (concept) lattice is built from a binary relation. It is used in machine learning to identify conceptual structures among data sets. A formal concept is a set of objects—extent—together with the set of attributes—intent—that describe each of the objects. For example, a set of documents and the set of search terms those documents match, form a concept.

Given the data in Fig. 2a, the smallest concept including document 1 is the one with documents {1, 2} and search terms {A,B,C,D}. Concepts can be partially ordered by inclusion and thus can be represented by a lattice as in Fig. 2b

Galois lattices are related to diamond cubes: a perfect 3×3 diamond is described in the central element of the lattice of Fig. 2b. Formal Concept Analysis is typically restricted to two dimensions although Cerf et al. [33] generalise formal concepts by presenting an algorithm that is applied to more than two dimensions. Their definition of a closed n -set—a formal concept in more than two dimensions—states that each element is related to all others in the set and no other element can be added to this set

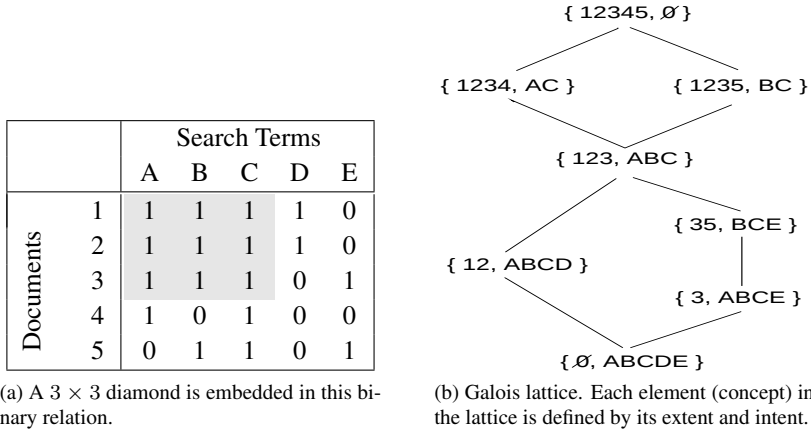


Figure 2: Documents and search terms in an information retrieval system and the corresponding Galois lattice.

without breaking the first condition. It is the equivalent of finding a perfect diamond in n dimensions. For example, given the data of Table 2 $\{(\alpha, \gamma)(1, 2)(A, B)\}$ is an example of a closed 3-set and also a perfect 4,4,4-diamond.

		dimension 3			dimension 3			dimension 3		
		A	B	C	A	B	C	A	B	C
dimension 2	1	1	1	1	1	1	1	1	1	
	2	1	1		1	1		1	1	
	3		1				1	1		1
	4			1	1		1	1	1	1
		α			β			γ		
		dimension 1								

Table 2: A 3-dimensional relation with closed 3-set $\{(\alpha, \gamma)(1, 2)(A, B)\}$.

2.4. Applications for Diamond Cubes

Diamonds have the potential to be useful in different areas and we present two specific applications as examples. First, we suggest they could be used when generating tag clouds—visual indexes that typically describe the content of a Web site. Tag clouds are defined and discussed in the next section. Second, one of the original motivations for this work was to identify a single large, dense subcube in any given data set. Once found, a dense region can be exploited to facilitate visualisation of the data or be used for further statistical analysis. Related work in this area is discussed in Section 2.4.2.

	Chicago	Montreal	Miami	Victoria	Saint John
Dog	4	2	2	1	1
Cat	2	4	1	6	1
Goldfish	10	3	4	1	3
Rabbit	3	11	1	2	1
Snake	1	1	1	1	9

Figure 3: Tags and their corresponding weights.

2.4.1. Visualisation

It is difficult to visualise large multidimensional data sets [9, 34, 35]. Aouiche et al. proposed tag clouds as an alternative [36]. Tag clouds are used on Web sites to communicate the relative importance of facts or topics discussed on a particular page. A tag is defined [36] as a term or phrase describing an object with corresponding non-negative weights determining its relative importance, and it comprises the triplet (term, object, weight).

For practical reasons, only the top values in each dimension are presented in a tag cloud. They can be selected by a top- k query on each dimension. However, if we seek significant values interacting together in a multidimensional way, then we can use the diamond operator. This would ensure that if the user slices on a value, some of the “top” values from other dimensions remain relevant. Consider Fig. 3 and its corresponding tag cloud. Taking the top-2 from each dimension the cloud comprises goldfish, rabbit, Chicago and Montreal.

If we slice on Chicago using a top-2 query, then *goldfish* and *dog* would be part of the cloud, which may very well confuse the user as *dog* was not part of the originally presented tags. If however, we slice on Montreal using a top-2 query we have *cat* and *rabbit*, also potentially confusing. Conversely, a 13×13 diamond retains the shaded items no matter which slice is chosen.

2.4.2. Dense Subcubes

One motivation for the diamond dice is to find a single, non-trivial dense region within a data cube. If the region has sufficient density, then hybrid storage becomes possible: the dense region can be stored as a multidimensional array, whereas the tuples outside of the dense region can be stored sparsely, e.g. as a list of locations. Another benefit of partitioning dense and sparse regions is that it enables visual representation of data to be enhanced. The dense regions can be represented close together on a single screen. Ben Messaoud et al. [9] employ Multiple Correspondence Analysis (MCA) to reorganise the cube so that dense subcubes are moved together for easy visualisation: A complete disjunctive table representing the entire cube is built and then MCA is applied to obtain factorial axes. Eigenvalues for each axis are computed and the user chooses a number of axes to visualise. We repeat the example from Ben Messaoud et al. in Fig. 4. As with diamond dicing, this approach measures the contribution of a set of cells to the overall cube. No timings were given in their paper, so it is difficult to compare the efficiency of MCA and diamond dicing, although the outputs are quite

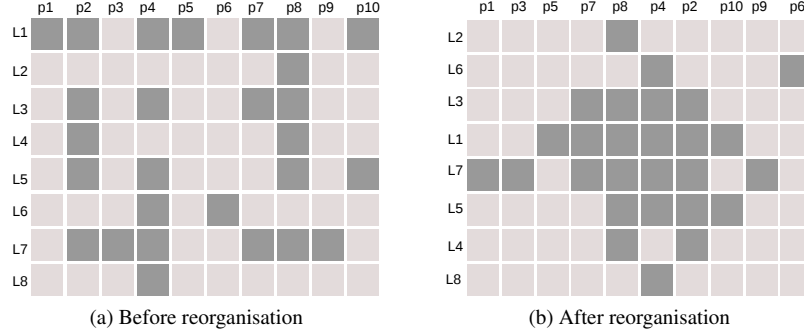


Figure 4: The allocated cells are in dark grey: they are moved together by the reorganisation.

similar.

Some researchers seek multiple dense regions rather than a single one, either to compute range sum queries more efficiently [37] or as a preprocessing step before further statistical analysis [38]. Lee uses a histogram-flattening technique by computing three relative density measures for each attribute in each dimension. Dense regions in each dimension are then used to build multi-dimensional dense regions. The candidate results are then refined iteratively. Experiments conducted on synthetic data seeded with clusters show that this method achieves good space reduction. However, there is no evidence of experimentation on real data or the time to compute their structure.

Barbará and Wu [38] seek to identify dense regions that can be further analysed by statistical methods. They shuffle attributes within dimensions based on a similarity measure. They do not mention whether rearranging attributes in one dimension has an adverse affect on the positioning of attributes in another dimension. Intuitively, one would expect there to be some issues with this process. The example provided in their paper identified four dense regions and showed how their algorithm would gather them together. The diamond dice applied to their example would find the largest dense region. If one then applied a diamond dice to the pruned area iteratively, all the regions identified by Barbará’s method would also be identified by the diamond dice for this particular example.

3. Properties of Diamond Cubes

In this section a formal model of the diamond cube is presented. We show that diamonds are nested, with a smaller diamond existing within a larger diamond. We prove a uniqueness property for diamonds and we establish upper and lower bounds on the parameter k for both COUNT and SUM-based diamond cubes.

3.1. Formal Model

Researchers and developers have yet to agree on a single multidimensional model for OLAP [39]. Our simplified formal model incorporates several widely-accepted def-

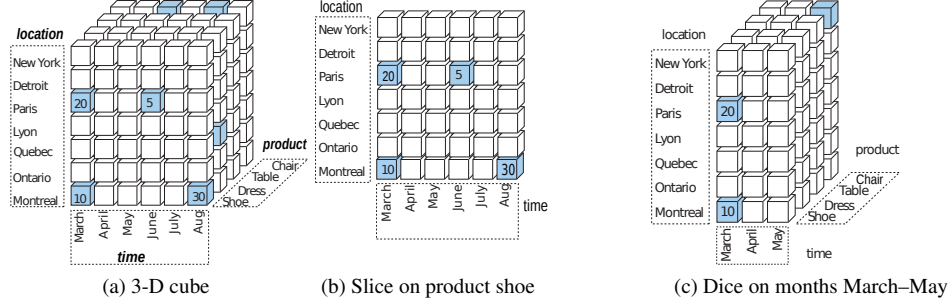


Figure 5: OLAP terms: cube, slice and dice.

initions for the terms illustrated in Fig. 5, together with new terms associated specifically with diamonds. For clarity, all terms are defined in the following paragraphs.

A *dimension* D is a set of *attributes* that defines one axis of a multidimensional data structure. For example, in Fig. 5 the dimensions are location, time and product. Each dimension D_i has a cardinality n_i , the number of distinct attribute values in this dimension. Without losing generality, we assume that $n_1 \leq n_2 \leq \dots \leq n_d$. A dimension can be formed from a single attribute of a database relation and the number of dimensions is denoted by d .

A *cube* is the 2-tuple (D, f) which is the set of dimensions $\{D_1, D_2, \dots, D_d\}$ together with a total function (f) which maps tuples in $D_1 \times D_2 \times \dots \times D_d$ to $\mathbb{R} \cup \{\perp\}$, where \perp represents undefined. Fig. 5a shows a *cube* with three dimensions.

A *cell* of cube C is a 2-tuple $((x_1, x_2, \dots, x_d) \in D_1 \times D_2 \times \dots \times D_d, v)$ where $v = f(x_1, x_2, \dots, x_d)$ is called a *measure*. The *measure* may be a value $v \in \mathbb{R}$, in which case we say the cell is an *allocated cell*. Otherwise, the measure is \perp and we say the cell is empty—an *unallocated cell*. For the purposes of this paper, a measure is a single value. In more general OLAP applications, a cube may map to several measures. Also, measures may take values other than real-valued numbers—booleans, for example.

A *slice* is the cube $C' = (D', f')$ obtained when a single attribute value is fixed in one dimension of cube $C = (D, f)$. For example, Fig. 5b is a slice of the cube presented in Fig. 5a.

A *dice* defines a cube S from an existing cube by removing attribute values and the corresponding cells. For example, Fig. 5c illustrates a dice applied to the cube from Fig. 5a where all months except March, April and May were removed. The resulting cube still has the same number of dimensions. We call it a *subcube* because its dimensions are subsets of the dimensions of the original cube, and, as a function, it is a restriction to the corresponding subset of cells.

An *aggregator* is a function, σ , that assigns a real number to a set of cells—such as a slice. For example, SUM is an aggregator: $\text{SUM}(\text{slice}_i) = v_1 + v_2 + \dots + v_m$ where m is the number of allocated cells in slice_i and the v_i 's are the measures.

A slice S' is a subset of slice S if every allocated cell in S' is also an allocated cell in S . An aggregator σ is monotonically non-decreasing if $S' \subset S$ implies $\sigma(S') \leq \sigma(S)$.

movie	reviewer	date	rating
1	1488844	2005-09-06	3
1	822109	2005-05-13	5
1	885013	2005-10-19	4
1	30878	2005-12-26	4
1	823519	2004-05-03	3
1	893988	2005-11-17	3
1	124105	2004-08-05	4
1	1248029	2004-04-22	3

Figure 6: Part of the Netflix [40] fact table (cube). Attributes (dimensions) are movie, reviewer and date. Each row is a fact (allocated cell). The measure is rating.

Similarly, σ is monotonically non-increasing if $S' \subset S$ implies $\sigma(S') \geq \sigma(S)$. Monotonically non-decreasing operators include COUNT, MAX and SUM over non-negative measures. Monotonically non-increasing operators include MIN and SUM over non-positive measures. MEAN and MEDIAN are neither monotonically non-increasing, nor non-decreasing functions.

Our formal model maps to the relational model in the following ways: (See Fig. 6.)

- A **cube** corresponds to a fact table: a relation whose attributes comprise a primary key and a single measure.
- An **allocated cell** is a fact, i.e. it is a distinct record in a fact table.
- A **dimension** is one of the attributes that compose the primary key.

3.2. Diamond Cubes are Unique

Intuitively, a diamond cube is a subcube where all attribute values satisfy a threshold condition. For example, all selected stores must have total sales over one million dollars. We call such conditions carats.

Definition 3.1. *Given a number k , a dimension has k carats if the aggregate of every slice along that dimension is at least k . That is, for every slice x , we have $\sigma(x) \geq k$.*

Note that if a dimension has k carats, it necessarily has k' carats for $k' \in [0, k)$. Given two subcubes A and B of the same starting cube, their union $A \cup B$ is defined by the union of the pairs of dimensions. For example, if A is the result of a dice limiting the location to Montreal and B is the result of a dice limiting the location to Toronto, the subcube $A \cup B$ will be the result of a dice limiting the location to both Montreal and Toronto. Similarly, the intersection $(A \cap B)$ is defined by the intersection of the pairs of dimensions. We say that subcube A is contained in subcube B if all of the dimensions of A are contained in the corresponding dimensions of B .

For monotonically non-decreasing operators (e.g., COUNT, MAX or SUM over non-negative measures), union preserves the carat, as the next proposition shows.

row	col 1	col 2	col 3	col 4	col 5	col 6
1	-5	1	1	1	0	3
2	-3	-4	1	0	1	0
3	2	2	4	0	2	1
4	0	2	3	1	0	0

(a) Cube with positive and negative measures.

row	col 2	col 3
3	2	4
4	2	3

(b) **rows** processed first.

row	col 3	col 6
1	1	2
4	4	2

(c) **columns** processed first.

Figure 7: There is no unique (4,4) diamond for this cube.

Proposition 3.1. *If the aggregator σ is monotonically non-decreasing, then the union of any two cubes having k_i (resp. k'_i) carats over dimension D_i has $\min(k_i, k'_i)$ carats over dimension D_i as well, for $i = \{1, 2, \dots, d\}$.*

Proof. The proof follows from the monotonicity of the aggregator. \square

If we limit ourselves to monotonically non-decreasing aggregators, then we can efficiently seek the largest possible subcube satisfying a given set of carats. We call such a subcube the **diamond**.

Definition 3.2. *The k_1, k_2, \dots, k_d -carat diamond is the maximal subcube having k_1, k_2, \dots, k_d carats over dimensions D_1, D_2, \dots, D_d . That is, any subcube having k_1, k_2, \dots, k_d carats is contained in the diamond.*

By Proposition 3.1, the diamond is unique when σ is monotonically non-decreasing: it is given by the union of all subcubes having k_1, k_2, \dots, k_d carats. For more general aggregators, the computation of the diamond might be NP-hard or ill-defined [41]. For instance, when SUM is used over cubes having both positive and negative measures, there may no longer be a *unique* solution to the problem ‘find the k_1, k_2, \dots, k_d -carat cube’. This is indeed the case for the cube in Fig. 7.

Any reference to a k -carat diamond cube implies $k_1 = k_2 = \dots = k_d = k$.

3.3. Diamond Cubes are Nested

The following proposition show that diamonds are nested. This is helpful because the x_1, x_2, \dots, x_d -carat diamond can be derived from the y_1, y_2, \dots, y_d -carat diamond when $y_i \leq x_i$ for all i .

Proposition 3.2. *The diamond having k' carats over dimensions i_1, \dots, i_d is contained in the diamond having k carats over dimensions i_1, \dots, i_d whenever $k' \geq k$.*

Proof. Let A be the diamond having k carats and B be the diamond having k' carats. By Proposition 3.1, $A \cup B$ has at least k carats, and because A is maximal, $A \cup B = A$; thus, B is contained in A . \square

4. A Priori Bounds on the Carats

The computations of a diamond requires that the analyst specifies the desired number of carats. However, this may not be practical for all dimensions. For example, the analyst may want to select stores with sales above one million dollars, but he may not know how to select the threshold for the product dimension. In such cases, it might be best to set the carats to the largest possible value which generates a non-empty diamond. Finding this maximal number of carats can be done efficiently by binary search if we can determine a limited range of possible values.

Given a cube C and σ , then κ is the largest number of carats for which the cube has a non-empty diamond. Intuitively, a small cube with many allocated cells should have a large κ , and the following proposition makes this precise.

Proposition 4.1. *For COUNT-based carats, we have $\kappa \geq |C| / \sum_i (n_i - 1) - 3$.*

Proof. Solving for k in Corollary A.1 (see Appendix A), we have a lower bound on the maximal number of carats: $\kappa \geq |C| / \sum_i (n_i - 1) - 3$. \square

Based on this lower bound alone, we compute κ efficiently (see § 6.4). For a related discussion on SUM-based diamonds, see Appendix B.

5. Algorithms

Computing diamonds is challenging because of the interaction between dimensions; modifications to a measure associated with an attribute value in one dimension have a cascading effect through the other dimensions. We use several different approaches to compute diamonds:

- Although there is no publicly available implementation of Kumar et al.’s algorithm [17], we have developed one that resembles the description provided in their paper. We refer to it as *Kumar* for the remainder of this paper. It has been extended to address more than two dimensions.
- We implemented a custom program in Java that loops through the cube checking and updating the COUNT or SUM for all attribute values in each dimension until it stabilises (see § sec:external).
- We also implemented an algorithm using SQL (see § 5.2).

We based both our custom and SQL implementations on the basic algorithm for computing diamonds given in Algorithm 1. Its approach is to repeatedly identify an attribute value that cannot be in the diamond, and then remove the attribute value and its slice. The identification of “bad” attribute values is done conservatively, in that they are known already to have a sum less than required (σ is SUM), or insufficient allocated cells (σ is COUNT). When the algorithm terminates, only attribute values that meet the condition in every slice remain: a diamond.

Algorithms based on this approach always terminate, though they might sometimes return an empty cube. By specifying *how* to compute and maintain sums for each attribute value in every dimension we obtain different variations. The correctness of any such variation is guaranteed by the following result.

```

input: a  $d$ -dimensional data cube  $C$ , a monotonically non-decreasing
        aggregator  $\sigma$  and  $k_1 \geq 0, k_2 \geq 0, \dots, k_d \geq 0$ 
output: the diamond cube  $A$ 
stable  $\leftarrow$  false
while  $\neg$ stable do
    // major iteration
    stable  $\leftarrow$  true
    for  $\text{dim} \in \{1, \dots, d\}$  do
        for  $i$  in all attribute values of dimension  $\text{dim}$  do
             $C_{\text{dim},i} \leftarrow \sigma(\text{slice } i \text{ on dimension } \text{dim})$ 
            if  $C_{\text{dim},i} < k_{\text{dim}}$  then
                delete attribute value  $i$ 
                stable  $\leftarrow$  false
            end
        end
    end
end
return cube without deleted attribute values;

```

Algorithm 1: Algorithm to compute the diamond of any given cube by deleting slices eagerly.

Theorem 5.1. *Algorithm 1 is correct, that is, it always returns the k_1, k_2, \dots, k_d -carat diamond.*

Proof. Because the diamond is unique, we need only show that the result of the algorithm, the cube A , is a diamond. If the result is not the empty cube, then dimension D_i has at least value k_i per slice, and hence it has k_i carats. We only need to show that the result of Algorithm 1 is maximal: there does not exist a larger k_1, k_2, \dots, k_d -carat cube.

Suppose A' is such a larger k_1, k_2, \dots, k_d -carat cube. Because Algorithm 1 begins with the whole cube C , there must be a first time when one of the attribute values of dimension dim of C belonging to A' but not A is deleted.

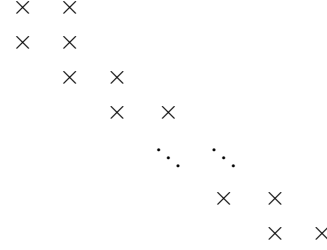
At the time of deletion, this attribute's slice cannot have obtained more cells so it still has value less than k_{dim} . Let C' be the cube at the instant before the attribute is deleted, with all attribute values deleted so far. We see that C' is larger than or equal to A' and therefore, slices in C' corresponding to attribute values of A' along dimension dim must have more than k_{dim} carats. Therefore, we have a contradiction and must conclude that A' does not exist and that A is maximal. \square

5.1. Custom Software

The size of available memory affects the capacity of in-memory data structures to represent data cubes. In our experiments we restricted the size of available memory to less than 2 GiB. We are interested in processing large data. Therefore, we seek an efficient external memory implementation where the data cube can be stored in an external file whilst the important counts (resp. sums) are maintained in memory.

cell	PID	SID	month
i	1	a	Jan
ii	1	b	Jan
iii	2	c	Feb
iv	2	d	Mar
v	3	a	Feb
vi	3	b	Feb
vii	3	d	Apr
viii	4	c	Apr
ix	4	e	Jan

(a) Computing the 2-carat diamond would require 3 iterations over this cube.



(b) An $n \times n$ cube with $2n$ allocated cells (each indicated by a \times) and a 2-carat diamond in the upper left: it is a difficult case for several algorithms.

Figure 8: Constructing an SQL query to compute diamonds is challenging

Algorithm 1 checks the σ -value for each attribute on every iteration. Calculating this value directly, from a data cube too large to store in main memory, would entail many expensive disk accesses. Even with the COUNTS maintained in main memory, it is prudent to reduce the number of I/O operations as much as possible. One way this can be achieved is to store the data cube as normalised binary integers using bit compaction [42]—mapping strings to small integers starting at zero.

Algorithm 2 employs d arrays, a_1 to a_d , that map attributes to their aggregate σ -values. As values are pruned from the diamond, we must repeatedly update these arrays so that they continue to maintain the aggregate of each slice. This update can be executed in constant time for aggregators such as COUNT and SUM: in the notation of Algorithm 2, the update is computed as $a_j(x_j) = a_j(x_j) - \sigma(\{r\})$.

Each time the algorithm passes through the data, it updates the aggregates eagerly, but only marks the cells as deleted. Only when a significant fraction of the cells have been marked as deleted are the cells actually deleted: we found that rebuilding the list of cells when more than half of the cells are marked as deleted to be efficient ($\tau = 0.5$). When memory is abundant, we can use Algorithm 2 while keeping the content of the files in memory.

5.2. An SQL-Based Implementation

A data cube can be represented in a relational database by a fact table. (See Fig. 8a.) In this example the dimensions are: productID {1,2,3,4}, salesPersonID {a,b,c,d,e} and month {Jan, Feb, Mar, Apr}. The cell numbers are noted in this example, but they do not form part of the cube.

Formulating a diamond cube query in SQL-92 is challenging since examples can be constructed where only one attribute ever has $\sigma(\text{slice}_{\text{dim}, i}) < k$, but its removal exposes another attribute, and so on. See Figure 8b and consider computing the 2-carat diamond from the cube in Fig. 8a. Every dimension must be examined three times to determine that cells i , ii , v and vi comprise the 2-carat diamond.

```

input: file inFile containing  $d$ -dimensional cube  $C$ , integers
         $k_1, k_2, \dots, k_d > 0$ , monotonically non-decreasing aggregator  $\sigma$ , a
        parameter  $\tau \in [0, 1)$ 
output: the cells in the diamond data cube
foreach dimension  $i$  do
    Create array  $a_i$  of size  $|D_i|$ 
    foreach attribute value  $v$  in dimension  $i$  do
         $a_i(v) = \sigma(\text{slice for value } v \text{ of dimension } i \text{ in } C)$ 
    end
end
stable  $\leftarrow$  false
while  $\neg$ stable do
    stable  $\leftarrow$  true
    foreach cell  $r$  of inFile which is not marked as deleted do
         $((x_1, x_2, \dots, x_d), v) \leftarrow r$ 
        Deleted  $\leftarrow$  false
        for  $i \in \{1, \dots, d\}$  do
            if  $a_i(x_i) < k_i$  then
                // attribute  $x_i$  had previously been deleted
                from the diamond
                for  $j \in \{1, \dots, i-1, i+1, \dots, d\}$  do
                    update  $a_j(x_j)$  given the removal of cell  $r$ 
                end
                stable  $\leftarrow$  false
                mark  $r$  as deleted
                break
                // only delete this cell once
            end
        end
    end
    if the fraction of cells marked as deleted exceeds  $\tau$  then
        rebuild inFile without the deleted cells
    end
end
rebuild inFile without the deleted cells
return inFile

```

Algorithm 2: Diamond dicing for relationally stored cubes. Each iteration, less data is processed.

Using nested queries and joins, we could essentially simulate a fixed number of iterations of the outer loop in Algorithm 1. Unfortunately, we cannot bound the number of iterations for that loop by a constant, leaving this approach as merely approximating the diamond. In Fig. 8b, we see an example where the number of iterations $I = \Omega(n)$ and stopping after $o(n)$ iterations results in a poor approximation with $\Theta(n)$ allocated cells and attribute values—whereas the true 2-diamond has 4 attribute values and 4 allocated cells.

INPUT: a d -dimensional data cube C and $k > 0$
OUTPUT: the diamond A
 initialise \mathcal{R} to C , the fact table
repeat {major iteration}
 execute the fragment of SQL pseudocode shown below
until no records were deleted from \mathcal{R}
 return \mathcal{R} as A

```
CREATE TABLE temp1 AS
(SELECT dim1 FROM  $\mathcal{R}$ 
 GROUP BY dim1 HAVING  $\sigma(measure) < k$ );

...
CREATE TABLE tempd AS
(SELECT dimd FROM  $\mathcal{R}$ 
 GROUP BY dimd HAVING  $\sigma(measure) < k$ );
DELETE FROM  $\mathcal{R}$ 
WHERE dim1 IN (SELECT * FROM temp1) OR ...
      dimd IN (SELECT * FROM tempd);
```

Algorithm 3: Variation where the inner two loops in Algorithm 1 are computed in SQL. This process can be repeated until \mathcal{R} stabilises.

We express the essential calculation in SQL, as Algorithm 3. It is implemented as a stored procedure, in SQL:1999, which allows the iterations to be controlled entirely within the DBMS. Algorithm 3 can be executed against a copy of the fact table, which becomes smaller as the algorithm progresses. The fastest variation of this algorithm does not delete slices immediately, but instead updates boolean values to indicate the slices are not included in the solution. The data cube is rebuilt when 75% of the remaining cells are marked for deletion. B-tree indexes are built on each dimension to facilitate faster execution of the many GROUP BY clauses.

5.3. Complexity Analysis

Algorithm 1 visits each dimension in sequence until it stabilises. Ideally, the stabilisation should occur after as few iterations as possible.

Let I be the number of iterations through the input file till convergence; i.e. no more deletions are done. Value I is data dependent and (by Fig. 8b) is $\Theta(\sum_i n_i)$ in the worst case. In practice, I is not expected to be nearly so large, and working with

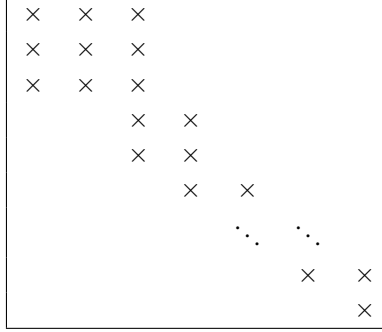


Figure 9: The 2-carat diamond requires more iterations to converge than 3-carat diamond.

large real data sets I did not exceed 56. Experimentally, it appears that the relationship of I to k is bi-tonic, i.e. it is non-decreasing to $\kappa + 1$ and non-increasing thereafter. Unfortunately, there are some cubes for which this is not the case. Fig. 9 illustrates such a cube. On the first iteration, processing columns first for the 2-carat diamond, a single cell is deleted. On subsequent iterations at most two cells are deleted until convergence. However, the 3-carat diamond converges after a single iteration.

The value of k relative to κ does, however, influence I . Typically, when k is far from κ —either less or greater—fewer iterations are required to converge. However, when k exceeds κ by a very small amount, say 1, then typically a great many more iterations are required to converge to the empty cube.

Algorithm 2 runs in time $O(Id|C|)$; each attribute value is deleted at most once. Often, the number of attribute values remaining in the diamond decreases substantially in the first few iterations and those cubes are processed faster than this bound suggests. The more carats we seek, the faster the cube decreases initially.

6. Experiments

We show that diamonds can be computed efficiently, i.e. within a few minutes and using less than 2 GiB of memory even for very large data sets. Some of the properties of diamonds, including their density (count-based diamonds) and the range of values the carats may take, were reviewed experimentally.

6.1. Hardware and Software

All experiments were conducted on a single Dell PE1950 compute node connected to a 16-node cluster through an Infiniband communications link. Each node contains 2 quad core Intel(R) Xeon(R) E5405 (2.00 GHz) processors with 8 GiB of RAM.

The algorithms were implemented in Java, using SDK version 1.6.0, and the code is archived at a public website [16]. The SQL experiments were conducted on MySQL version 5.5 Community Server with MyISAM storage engine and making use of a stored procedure. A Java interface connected to the database with mysql-connector-java-5.1.15 to collect execution times. Of the index structures available in this version

Table 3: Statistics of TWEED, Netflix, census and weather data.

cube	dimensions	$ C $	$\sum_{i=1}^d n_i$	measure
TW1	4	1 957	88	count
TW2	15	4 963	674	count
TW3	15	4 963	674	sum killed
NF1	3	100 478 158	500 137	count
NF2	3	100 478 158	500 137	sum rating
NF3	4	20 000 000	473 753	count
C1	27	135 753	5 607	count
C2	27	135 753	504	sum stocks
W1	11	124 164 371	48 654	count
W2	11	124 164 371	48 654	sum total cloud cover

Table 4: Statistics of the King James Bible data.

cube	dimensions	$ C $	$\sum_{i=1}^d n_i$	measure
B1	4	54 601 077	404	count
B2	4	24 000 000	325	count
B3	4	32 000 000	322	count
B4	4	40 000 000	364	count
B5	4	54 601 077	303	sum 4-gram instances
B6	10	365 231 367	96	count

of MySQL, only B-trees are appropriate to the diamond dice operation. Spatial indexing is limited to two dimensions and hash indexing requires that the data reside in main memory.

6.2. Data Used in Experiments

A varied selection of freely-available real-data sets together with some systematically generated synthetic data sets were used in the experiments. Each data set had a particular characteristic: a few dimensions or many, dimensions with high or low cardinality or a mix of the two, small or large number of cells. They were chosen to illustrate that diamond dicing is tractable under varied conditions and on many different types of data.

6.2.1. Real Data

Four of the real-data sets were downloaded from the following sources:

1. TWEED: <http://folk.uib.no/sspje/tweed.htm> [43]
2. Netflix x: <http://www.netflixprize.com> [44]
3. Census Income: <http://kdd.ics.uci.edu/> [45]
4. Weather: <http://cdiac.ornl.gov/ftp/ndp026b/> [46]

Table 5: Statistics of the synthetic data cubes.

Cube	dimensions	$ C $	$\sum_i n_i$
U1	3	999 987	10 773
U2	4	1 000 000	14 364
U3	10	1 000 000	35 910
S1	3	939 153	10 505
S2	4	999 647	14 296
S3	10	1 000 000	35 616
SS1	3	997 737	74 276
SS2	4	999 995	99 525
SS3	10	1 000 000	248 703

Details of how the cubes were extracted are available at a public website [16] and their statistics are given in Table 3. Each cube was stored relationally in a comma-separated file on disk. A brief description of how data cubes were extracted from the King James Bible data follows.

The fifth data set was generated from the King James version of the Bible available at Project Gutenberg [47]. KJV-4grams [48, 49] is a data set motivated by applications of data warehousing to literature. It is a large list (with duplicates) of 4-tuples of words obtained from the verses in the King James Bible [47], after stemming with the Porter algorithm [50] and removal of stemmed words with three or fewer letters. Occurrence of row w_1, w_2, w_3, w_4 indicates a verse contains words w_1 through w_4 , in this order. This data is a scaled-up version of word co-occurrence cubes used to study analogies in natural language [51, 52]. These data were chosen to be representative of large cubes that might occur in text mining applications.

B1 was extracted from KJV-4grams. Duplicate records were removed and a count of each unique sequence was kept, which became the measure for cube B5. Four sub-cubes of B1 were also processed: B2: first 24 000 000 rows; B3: first 32 000 000 rows; and B4: first 40 000 000 rows. KJV-10grams has similar properties to KJV-4grams, except that there are 10 words in each row and the process of creating KJV-10grams was terminated when 500 million records had been generated—at the end of Genesis 19:30. Cube B6 was extracted from KJV-10grams. The statistics for all six cubes are given in Table 4.

6.2.2. Synthetic Data

To investigate the effect that data distribution might have on the size and shape of diamonds, nine cubes of varying dimensionality and distribution were constructed. We chose 1 000 000 cells with replacement from each of three different distributions:

- uniform—cubes U1, U2, U3.
- power law with exponent 3.5 to model the 65-35 skewed distribution—cubes S1, S2, S3.

Table 6: Wall-clock times (in seconds) for preprocessing real-world data sets. DB-Count and DBSum implement Algorithm 2 for both string and binary data. A ‘—’ indicates that this algorithm was not applied to the corresponding data cube.

Cube	DBCount/DBSum (Alg. 2)	Kumar	SQL (Alg. 3)
B1	2.9×10^2	—	—
B2	8.4×10^1	—	1.5×10^3
B3	1.1×10^2	—	2.0×10^3
B4	1.4×10^2	—	2.6×10^3
C1	7×10^0	1.4×10^2	6×10^0
NF1	2.8×10^2	3.7×10^3	3.6×10^3
NF3	9.1×10^1	1.0×10^3	1.3×10^3
TW1	8×10^{-2}	1.8×10^{-1}	1.8×10^{-1}

- power law with exponent 2.0 to model the 80-20 skewed distribution—cubes SS1, SS2, SS3.

Details of the cubes generated are given in Table 5.

6.3. Preprocessing Step

Before applying Algorithm 2, we need to convert the input flat text files to flat binary files. To determine if row ordering would have an effect on our binary implementation of Algorithm 2, we chose two cubes—C1 and B2—and shuffled the rows using the Unix utility *shuf*. We compared preprocessing and processing times for each of six cubes, averaged over ten runs. Extracting cubes from the data sets included a sorting step so that duplicates could be easily removed. We found that preprocessing the cube sorted on its dimension of largest cardinality was up to 25% faster than preprocessing the shuffled cube. However, execution times for Algorithm 2 were within 3% for each cube. Therefore, we did not reorder the rows prior to processing. We also found no significant difference in execution times when the cubes were sorted by different dimensions.

The algorithms used in our experiments required different preprocessing of the cubes. For Algorithm 2 an in-memory data structure was used to maintain counts of the attribute values. The Kumar algorithm used d sorted files and Algorithm 3 referenced the cube stored in a relational database. Consequently, the preprocessor wrote different kinds of data to supplementary files depending on which algorithm was to be used.

The preprocessing of the cubes was timed separately from diamond building. Preprocessed data could be used many times, varying the value for k , without incurring additional preparation costs. Table 6 summarises the times needed to preprocess each cube in preparation for the algorithms that were run against it. For comparison, sorting the Netflix comma-separated data file—using the GNU sort utility—took 18 minutes. The Kumar algorithm did not work well with high dimensional cubes. It requires a sorted copy of the cube for each dimension which incurs additional space and I/O costs. We only ran it against a few cubes.

Table 7: Iterations to convergence for sum and count diamonds

(a) The number of iterations it took to determine the κ -diamond for COUNT-based diamonds.

cube	iters	value of κ	
		est.	actual
TW1	6	19	38
NF1	19	197	1 004
NF3	17	39	272
C1	8	282	672
W1	26	2 550	4 554
B1	12	1 723	14 383
B2	16	884	7 094
B3	19	1 098	8 676
B4	12	1 347	10 513
B6	5	57 668	112 232 566

(b) The number of iterations it took to determine the κ -diamond on SUM-based diamonds. The estimate for κ is the tight lower bound from Proposition B.1.

cube	iters	value of κ	
		est.	actual
TW3	3	85	85
NF2	40	5	3 483
C2	5	1 853	3 600 675
W2	19	32	20 103
B5	4	729	25 632

6.4. Finding κ for COUNT-based Diamonds

Using Proposition 4.1, the κ -diamond was built for each of the data sets. The initial guess (k) for κ was the value calculated using Proposition 4.1. When a non-empty diamond was built, and for every data set this was indeed the case, k was repeatedly doubled until an empty cube was returned and a tighter range for κ had been established². Next a simple binary search, which used the newly discovered lower and upper bounds as the end points of the search space, was executed. Each time a non-empty diamond was returned, it was used as the input to the next iteration of the search. When the guess overshoot κ and an empty diamond was returned, the most recent non-empty cube was used as the input.

Statistics are provided in Table 7a. The estimate of κ comes from Proposition 4.1 and the number of iterations recorded is the number needed to find the κ -diamond. The estimates for κ varied between 4% and 50% of the actual value and there is no clear pattern to indicate why this might be. Two very different cubes both had estimates that were 50% of the actual value: TW1, a small cube of less than 2 000 cells and low dimensionality, and W1, a large cube of 123×10^6 cells with moderate dimensionality.

6.5. Finding κ for SUM-based Diamonds

From Proposition B.2, we have that $\min_i(\max_j(\sigma(\text{slice}_j(D_i))))$ is an upper bound for κ of any sum-diamond and from Proposition B.1 a lower bound is the maximum value stored in any cell. Indeed, for cube TW3 the lower bound is the κ value. For this

²This method was more efficient than using the theoretical upper bound on κ established in Proposition A.1.

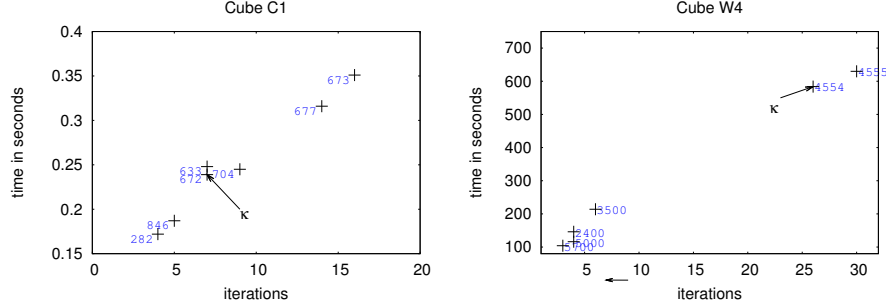


Figure 10: Times and iterations needed to generate diamonds with different k -values (labelled). In each case more time and iterations are required for k -values that slightly exceed κ .

reason, the approach to finding κ for the sum-diamonds varies slightly in that the first guess for k should be the lower bound + 1. If this returns a non-empty diamond, then a binary search over the range [lower bound + 1–upper bound] is used to find κ . Statistics are given in Table 7b.

6.6. Comparison of Algorithm Speeds

Neither the initial file size nor the number of cells pruned from each k -diamond alone explains the time necessary to generate each diamond.

In an earlier implementation of the diamond dicing algorithm [41], we had observed that the time expended was proportional to the number of cells processed. This is not as evident in the current implementation, where a new file is written when 50% of the cells are marked for deletion, instead of at every iteration.

Table 8 compares the speeds of Algorithm 3 (SQL4) and the binary version of Algorithm 2 (BIN) run against some of the data sets. Times were averaged over five runs and then normalised against the binary version. Two versions of BIN were implemented: each maps the data cube into memory. Whenever the data cube is small enough (< 500 Mb) we use our in-memory approach. For the larger cubes, an external-memory approach, which maps chunks of the cube into memory, is used. We see that our binary implementation effects greater speed-up as the cube size increases and the cube density decreases. For example, we see that BIN is 146 times faster than SQL4 on cube B2, but 498 times faster on the more sparse cube, NF3.

6.7. Diamond Size and Dimensionality

The size (in cells) of the κ -diamond of the high-dimensional cubes is large, e.g. the κ -diamond for B6 captures 30% of the data. How can we explain this? Is this property a function of the number of dimensions? To answer this question the κ -diamond was generated for each of the synthetic cubes. Estimated κ , its real value and the size in cells for each cube are given in Table 9. The κ -diamond captures 99% of the data in cubes U1, U2 and U3—dimensionality has no effect on diamond size for these uniformly

Table 8: Relative slowdown of the SQL algorithm compared to the binary implementation. Times were averaged over five runs and then normalised against the binary execution times. SQL processing for NF1 was forcibly terminated after 19 hours (\otimes).

Cube	BIN (s)	SQL4 (s)	SQL4 / BIN
TW1	0.01	0.04	4
C1	0.37	23.0	62
B2	36	5 291	146
B3	49	7 989	163
B4	57	10 553	183
NF3	10	4 988	498
NF1	124	\otimes	\otimes

Table 9: High dimensionality does not affect diamond size.

Cube	dimensions	iters	value of κ		size (cells)	% captured
			est.	actual		
U1	3	6	89	236	982 618	98
U2	4	6	66	234	975 163	98
U3	10	7	25	229	977 173	98
S1	3	9	90	1141	227 527	24
S2	4	14	67	803	231 737	23
S3	10	14	25	208	260 864	26
SS1	3	18	11	319	122 878	12
SS2	4	19	7	175	127 960	13
SS3	10	17	1	28	165 586	17

distributed data sets. Likewise, dimensionality did not affect the size of the κ -diamond for the skewed data cubes as it captured between 23% and 26% of the data in cubes S1, S2 and S3 and between 12% and 16.5% in the other cubes. These results indicate that the dimensionality of the cube does not affect how much of the data is captured by the diamond dice.

6.8. Iterations to Convergence

In Section 5.3 we observed that in the worst case it could take $\Theta(\sum_i n_i)$ iterations before the diamond cube stabilised. In practice this was not the case. (See Table 7a, Table 7b and Fig. 10.) All cubes converged to the κ -diamond in less than 1% of $\sum_i n_i$, with the exception of the small cube TW1, which took less than 7% $\sum_i n_i$. Algorithm 2 required 19 iterations and 2 minutes³ to compute the 1 004-carat κ -diamond for NF1 and it took 50 iterations and an average of 3 minutes to determine that there was no

³averaged over 10 runs

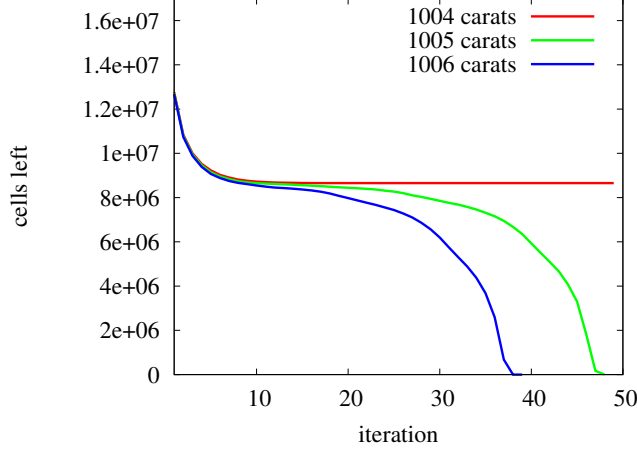


Figure 11: Cells remaining after each iteration of Algorithm 2 for $k = 1004, 1005$ and 1006 on cube NF1

1 005-carat diamond. We measured the number of cells remaining in cube NF1 after each iteration of different k -diamonds to determine how quickly the diamond converges to an empty diamond when k exceeds κ . Fig. 11 shows the number of cells present in the diamond after each iteration for 1 004–1 006 carats. The curve for 1 006 reaches zero first, followed by that for 1 005. Since $\kappa(\text{NF1}) = 1\,004$, that curve stabilises at a nonzero value. It takes longer to reach a critical point when k only slightly exceeds κ .

Intuitively, one should not be surprised that more iterations are required when $k \approx \kappa$: attribute values that are almost in the diamond are especially sensitive to other attribute values that are also almost in the diamond.

The number of iterations required until convergence for all our synthetic cubes was also far smaller than the upper bound, e.g. cube S3: 35 616 (upper bound) and 14 (actual). We had expected to see the uniformly distributed data taking longer to converge than the skewed data. This was not the case: in fact the opposite behaviour was observed. (See Table 9.) For cubes U1, U2 and U3 the diamond captured 98% of the cube: less than 23 000 cells were removed, suggesting that they started with a structure very like a diamond but for the skewed data cubes—S1, S2, S3, SS1, SS2 and SS3—the diamond was more “hidden”.

7. Conclusion

We presented a formal analysis of the diamond cube. We have shown that, for the parameter k associated with each dimension in every data cube, there is only one k_1, k_2, \dots, k_d -diamond. By varying the k_i ’s we get a collection of diamonds for a cube. We established upper and lower bounds on the parameter k for both COUNT and SUM-based diamond cubes.

We have designed, implemented and tested algorithms to compute diamonds on real and synthetic data sets. Experimentally, the algorithms bear out our theoretical results. An unexpected experimental result is that the number of iterations required to process the diamonds with k slightly greater than κ is often twice that required to process the κ -diamond, which also resulted in an increase in running time.

We have shown that computing diamonds for large data sets is feasible. Our binary implementation fared better on large, sparse data cubes than other approaches and these results confirm that this algorithm is scalable.

7.1. Future Research Directions

Although it is faster to compute a diamond cube using our implementation than using the standard relational DBMS operations or OLAP operators, the speed does not conform to the OLAP goal of near constant time query execution. Different approaches could be taken to improve execution speed: compress the data so that more of the cube can be retained in memory; use multiple processors in parallel; or, if an approximate solution is sufficient, we might process only a sample of the data. These are some of the ideas to be explored in future work.

Data cubes are often organised with hierarchies of relationships within dimensions. For example, a *time* dimension may include aggregations for year, month and day. Our current work does not address the issue of hierarchies and how they might be exploited in the computation of diamonds. This is also a potential avenue for future work.

References

- [1] D. N. Politis, J. P. Romano, M. Wolf, *Subsampling*, Springer, 1999.
- [2] B. Babcock, S. Chaudhuri, G. Das, Dynamic sample selection for approximate query processing, in: *SIGMOD'03*, pp. 539–550.
- [3] V. Ganti, M. L. Lee, R. Ramakrishnan, ICICLES: Self-tuning samples for approximate query answering, in: *VLDB'00*, pp. 176–187.
- [4] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, J. D. Ullman, Computing iceberg queries efficiently, in: *VLDB'98*, pp. 299–310.
- [5] J. Pei, M. Cho, D. Cheung, Cross table cubing: Mining iceberg cubes from data warehouses, in: *SDM'05*, pp. 461–465.
- [6] D. Xin, J. Han, X. Li, B. W. Wah, Star-cubing: Computing iceberg cubes by top-down and bottom-up integration, in: *VLDB'03*, VLDB Endowment, 2003, pp. 476–487.
- [7] Z. X. Loh, T. W. Ling, C. H. Ang, S. Y. Lee, Adaptive method for range top-k queries in OLAP data cubes, in: *DEXA'02*, pp. 648–657.
- [8] Z. X. Loh, T. W. Ling, C. H. Ang, S. Y. Lee, Analysis of pre-computed partition top method for range top-k queries in OLAP data cubes, in: *CIKM'02*, pp. 60–67.

- [9] R. Ben Messaoud, O. Boussaid, S. Loudcher Rabaséda, Efficient multidimensional data representations based on multiple correspondence analysis, in: KDD'06, pp. 662–667.
- [10] M. J. Carey, D. Kossmann, On saying “enough already!” in SQL, in: SIGMOD'97, pp. 219–230.
- [11] G. Cormode, S. Muthukrishnan, What's hot and what's not: Tracking most frequent items dynamically, *ACM Trans. Database Syst.* 30 (2005) 249–278.
- [12] G. Cormode, F. Korn, S. Muthukrishnan, D. Srivastava, Diamond in the rough: Finding hierarchical heavy hitters in multi-dimensional data, in: SIGMOD '04, ACM Press, New York, NY, USA, 2004, pp. 155–166.
- [13] H. Webb, Properties and applications of diamond cubes, in: ICSOFT 2007 – Doctoral Consortium.
- [14] H. Webb, O. Kaser, D. Lemire, Pruning attribute values from data cubes with diamond dicing, in: International Database Engineering and Applications Symposium (IDEAS'08), pp. 121–129.
- [15] M. Ley, Digital bibliography and library project, <http://dblp.uni-trier.de/xml/> (checked 2011-03-03), 2011.
- [16] H. Webb, Code archive, <http://www.hazel-webb.com/archive.htm>, 2009. (Last checked 06-09-2010).
- [17] R. Kumar, P. Raghavan, S. Rajagopalan, A. Tomkins, Trawling the Web for emerging cyber-communities, in: WWW '99, Elsevier North-Holland, Inc., New York, NY, USA, 1999, pp. 1481–1493.
- [18] P. K. Reddy, M. Kitsuregawa, An approach to relate the web communities through bipartite graphs, in: WISE'01, pp. 302–310.
- [19] K. Yang, Information retrieval on the web, *Annual Review of Information Science and Technology* 39 (2005) 33–81.
- [20] S. Raghavan, H. Garcia-Molina, Representing Web graphs, in: ICDE'03, ACM Press, 2003, pp. 1–10.
- [21] K. Holzapfel, S. Kosub, M. G. Maaß, H. Täubig, The complexity of detecting fixed-density clusters, 2006.
- [22] L. Tang, H. Liu, Graph mining applications to social network analysis, in: C. C. Aggarwal, H. Wang (Eds.), *Managing and Mining Graph Data*, volume 40 of *Advances in Database Systems*, Springer US, 2010, pp. 487–513.
- [23] X. Zhang, Y. Li, W. Liang, C&C: an effective algorithm for extracting web community cores, in: Proceedings of the 15th international conference on Database systems for advanced applications, DASFAA'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 316–326.

- [24] S. Börzsönyi, D. Kossmann, K. Stocker, The Skyline operator, in: ICDE '01, IEEE Computer Society, 2001, pp. 421–430.
- [25] D. Donjerkovic, R. Ramakrishnan, Probabilistic optimization of top n queries, in: VLDB'99, pp. 411–422.
- [26] M. L. Yiu, N. Mamoulis, Efficient processing of top-k dominating queries on multi-dimensional data, in: VLDB'07, pp. 483–494.
- [27] F. Korn, S. Muthukrishnan, Influence sets based on reverse nearest neighbor queries, in: Proceedings of the 2000 ACM SIGMOD international conference on Management of data, SIGMOD '00, ACM, New York, NY, USA, 2000, pp. 201–212.
- [28] P. Indyk, R. Motwani, P. Raghavan, S. Vempala, Locality-preserving hashing in multidimensional spaces, in: STOC '97, ACM, New York, NY, USA, 1997, pp. 618–625.
- [29] W. Dong, C. Moses, K. Li, Efficient k-nearest neighbor graph construction for generic similarity measures, in: Proceedings of the 20th international conference on World wide web, WWW '11, ACM, New York, NY, USA, 2011, pp. 577–586.
- [30] E. M. Knorr, R. T. Ng, Algorithms for mining distance-based outliers in large datasets, in: Proceedings of the 24rd International Conference on Very Large Data Bases, VLDB '98, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998, pp. 392–403.
- [31] R. Wille, Knowledge acquisition by methods of formal concept analysis, in: Proceedings of the Conference on Data Analysis, Learning Symbolic and Numeric Knowledge, Nova Science Publishers, Inc., Commack, NY, USA, 1989, pp. 365–380.
- [32] R. Godin, R. Missaoui, H. Alaoui, Incremental concept formation algorithms based on Galois (concept) lattices, Computational Intelligence 11 (1995) 246–267.
- [33] L. Cerf, J. Besson, C. Robardet, J.-F. Boulicaut, Closed patterns meet n-ary relations, ACM Trans. Knowl. Discov. Data 3 (2009) 1–36.
- [34] A. Maniatis, P. Vassiliadis, S. Skiadopoulos, Y. Vassiliou, G. Mavrogonatos, I. Michalarias, A presentation model & non-traditional visualization for OLAP, International Journal of Data Warehousing and Mining 1 (2005) 1–36.
- [35] K. Techapichetvanich, A. Datta, Interactive visualization for OLAP, in: ICCSA '05, pp. 206–214.
- [36] K. Aouiche, D. Lemire, R. Godin, Web 2.0 OLAP: From data cubes to tag clouds, Lecture Notes in Business Information Processing 18 (2009) 51–64.

- [37] S.-L. Lee, An effective algorithm to extract dense sub-cubes from a large sparse cube., in: A. M. Tjoa, J. Trujillo (Eds.), DaWaK, volume 4081 of *Lecture Notes in Computer Science*, Springer, 2006, pp. 155–164.
- [38] D. Barbará, X. Wu, Finding dense clusters in hyperspace: An approach based on row shuffling., in: *Advances in Web-age Information Management*, pp. 305–316.
- [39] S. Rizzi, A. Abelló, J. Lechtenbörger, J. Trujillo, Research in data warehouse modeling and design: Dead or alive?, in: DOLAP '06, ACM, New York, NY, USA, 2006, pp. 3–10.
- [40] J. Bennett, S. Lanning, The Netflix prize, in: *KDD Cup and Workshop 2007*.
- [41] H. Webb, Properties and Applications of Diamond Cubes, Ph.D. thesis, University of New Brunswick Saint John, 2010.
- [42] W. Ng, C. Ravishankar, Block-oriented compression techniques for large statistical databases, *IEEE Transactions on Knowledge and Data Engineering* 9 (1997) 314–328.
- [43] J. O. Engene, Five decades of terrorism in Europe: The TWEED dataset, *Journal of Peace Research* 44 (2007) 109–121.
- [44] Netflix, Inc., Nexflix prize, <http://www.netflixprize.com>(checked 05-23-2011), 2007.
- [45] S. Hettich, S. D. Bay, The UCI KDD archive, <http://kdd.ics.uci.edu> (checked 05-23-2011), 2000.
- [46] C. Hahn, S. Warren, J. London, Edited synoptic cloud reports from ships and land stations over the globe, 1982–1991, <http://cdiac.ornl.gov/ftp/ndp026b/> (checked 05-23-2011), 2004.
- [47] Project Gutenberg Literary Archive Foundation, Project Gutenberg, <http://www.gutenberg.org/> (checked 05-23-2011), 2009.
- [48] O. Kaser, D. Lemire, K. Aouiche, Histogram-aware sorting for enhanced word-aligned compression in bitmap indexes, in: DOLAP '08.
- [49] D. Lemire, O. Kaser, Reordering columns for smaller indexes, *Information Sciences* 181 (2011) 2550–2570.
- [50] M. F. Porter, An algorithm for suffix stripping, in: *Readings in information retrieval*, Morgan Kaufmann, 1997, pp. 313–316.
- [51] P. D. Turney, M. L. Littman, Corpus-based learning of analogies and semantic relations, *Machine Learning* 60 (2005) 251–278.
- [52] O. Kaser, S. Keith, D. Lemire, The LitOLAP project: Data warehousing with literature, in: CaSTA'06.

Appendix A. Properties of COUNT-based Diamonds

Proposition A.1. *Given the sizes of dimensions of a COUNT-based diamond cube, n_i , an upper bound for the number of carats k of a non-empty subcube is $\prod_{i=1}^{d-1} n_i$. An upper bound on the number of carats k_i for dimension i is $\prod_{j=1, j \neq i}^d n_j$.*

Proof. Consider a *perfect* diamond, i.e. all cells are allocated. We can extract a slice from this diamond by holding a value in some dimension, D_x , constant. The number of allocated cells in this slice will be the product of cardinalities of the remaining dimensions. If we choose D_x to be the dimension of largest cardinality, then the number of allocated cells in this slice is the carat (k) count for this diamond. Since all cells are allocated, there is no non-empty diamond with more than k carats. The rest of the result follows similarly. \square

An alternate upper bound on the number of carats in any dimension is $|C|$, the number of allocated cells in the cube. For sparse cubes, this bound may be more useful than that from Proposition A.1.

Intuitively, a cube with many carats needs to have many allocated cells: accordingly, the next proposition provides a lower bound on the size of the cube given the number of carats.

Proposition A.2. *For $d > 1$ and $\sigma = \text{COUNT}$, the size $|C|$ of a d -dimensional non-empty cube C of k carats satisfies $|C| \geq kn_d \geq k^{d/(d-1)}$; more generally, a non-empty k_1, k_2, \dots, k_d -carat cube has size satisfying*

$$|C| \geq \max k_i n_i \geq \left(\prod k_i \right)^{1/(d-1)}, i \in \{1, 2, \dots, d\}.$$

Proof. Pick dimension D_d : the cube has n_d slices along this dimension, each with k allocated cells, proving that $|C| \geq kn_d$.

We have that $kn_d \geq k(\sum_i n_i)/d$ —since the average number of allocated cells per slice must be less than or equal to $\max_{i \in \{1, 2, \dots, d\}} n_i$ —so that $|C|$ is at least $k(\sum_i n_i)/d$. If we prove that $\sum_i n_i \geq dk^{1/(d-1)}$ then we will have that $k(\sum_i n_i)/d \geq k^{d/(d-1)}$.

This result can be shown using Lagrange multipliers. Consider the problem of minimising $\sum_i n_i$ given the constraints $\prod_{i=1, 2, \dots, j-1, j+1, \dots, d} n_i \geq k$ for $j = 1, 2, \dots, d$. These constraints are necessary since all slices must contain at least k cells. The corresponding Lagrangian is $L = \sum_i n_i + \sum_j \lambda_j (\prod_{i=1, 2, \dots, j-1, j+1, \dots, d} n_i - k)$. By inspection, the derivatives of L with respect to n_1, n_2, \dots, n_d are zero and all constraints are satisfied when $n_1 = n_2 = \dots = n_d = k^{1/(d-1)}$. For these values, $\sum_i n_i = dk^{1/(d-1)}$ and this must be a minimum, proving the result. The more general result follows similarly, by proving that the minimum of $\sum n_i k_i$ is reached when $n_i = (\prod_{i=1, \dots, d} k_i)^{1/(d-1)} / k_i$ for all i 's. \square

We calculate the *volume* of a cube C as $\prod_{i=1}^d n_i$ and its *density* is the ratio of allocated cells, $|C|$, to the volume ($|C| / \prod_{i=1}^d n_i$). We can exploit the relationship between the size of a diamond and its density to determine whether a diamond of particular carat exists in a given data cube.

Theorem A.1. *For COUNT-based carats, if a cube does not contain a non-empty k -carat subcube, then it has at most $1 + (k - 1) \sum_{i=1}^d (n_i - 1)$ allocated cells. Hence, it has density at most $(1 + (k - 1) \sum_{i=1}^d (n_i - 1)) / \prod_{i=1}^d n_i$. More generally, a cube that does not contain a non-empty k_1, k_2, \dots, k_d -carat subcube has size at most $1 + \sum_{i=1}^d (k_i - 1)(n_i - 1)$ and density at most $(1 + \sum_{i=1}^d (k_i - 1)(n_i - 1)) / \prod_{i=1}^d n_i$.*

Proof. Suppose that a cube of dimension at most $n_1 \times n_2 \times \dots \times n_d$ contains no k -carat diamond. Then one slice must contain at most $k - 1$ allocated cells. Remove this slice. The amputated cube must not contain a k -carat diamond. Hence, it has one slice containing at most $k - 1$ allocated cells. Remove it. This iterative process can continue at most $\sum_i (n_i - 1)$ times before there is at most one allocated cell left: hence, there are at most $(k - 1) \sum_i (n_i - 1) + 1$ allocated cells in total. The more general result follows similarly. \square

Corollary A.1 follows from Theorem A.1:

Corollary A.1. *A cube of size greater than $1 + (k - 1) \sum_{i=1}^d (n_i - 1)$ allocated cells, that is, having density greater than*

$$\frac{1 + (k - 1) \sum_{i=1}^d (n_i - 1)}{\prod_{i=1}^d n_i},$$

must contain a non-empty k -carat subcube. If a cube contains more than $1 + \sum_{i=1}^d (k_i - 1)(n_i - 1)$ allocated cells, it must contain a non-empty k_1, k_2, \dots, k_d -carat subcube.

Proof. From Theorem A.1 we have that a cube cannot have more than $1 + \sum_{i=1}^d (k_i - 1)(n_i - 1)$ allocated cells unless it also contains a non-empty k -diamond. Therefore, if a cube has more than $1 + \sum_{i=1}^d (k_i - 1)(n_i - 1)$ allocated cells, it must contain a nonempty k -diamond. \square

Appendix B. Bounding the carats for SUM-based Diamonds

For SUM-based diamonds, the goal is to capture a large fraction of the sum. The statistic, κ , of a SUM-based diamond is the largest sum for which there exists a non-empty diamond: every slice in every dimension has sum at least κ (see § 4). Propositions B.1 and B.2 give tight lower and upper bounds respectively for κ .

Proposition B.1. *Given a non-empty cube C and the aggregator SUM, a tight lower bound on κ is the value of the maximum cell (m).*

Proof. The κ -diamond, by definition, is non-empty, so it follows that when the κ -diamond comprises a single cell, then κ takes the value of the maximum cell in C . When the κ -diamond contains more than a single cell, m is still a lower bound: either κ is greater than or equal to m . \square

Given only the size of a SUM-based diamond cube (in cells), there is no upper bound on its number of SUM-carats. However, given its sum, say S , then it cannot have more than S SUM-carats. We can determine a tight upper bound on κ as the following proposition shows.

Proposition B.2. *A tight upper bound for κ is*

$$\min_i (\max_j (\text{SUM}(\text{slice}_j(D_i)))) \text{ for } i \in \{1, 2, \dots, d\} \text{ and } j \in \{1, 2, \dots, n_i\}.$$

Proof. Let $X = \{\text{slice}_j(D_i) \mid \text{SUM}(\text{slice}_j(D_i)) = \max_k (\text{SUM}(\text{slice}_k(D_i)))\}$ then there is one slice x whose $\text{SUM}(x)$ is smaller than or equal to all other slices in X . Suppose κ is greater than $\text{SUM}(x)$ then it follows that all slices in this κ -diamond must have SUM greater than $\text{SUM}(x)$. However, x is taken from X , where each member is the slice for which its SUM is maximum in its respective dimension, thereby creating a contradiction. Such a diamond cannot exist. Therefore, $\min_i (\max_j (\text{SUM}(\text{slice}_j(D_i))))$ is an upper bound for κ .

To show that $\min_i (\max_j (\text{SUM}(\text{slice}_j(D_i))))$ is also a tight upper bound we only need to consider a perfect cube where all measures are identical. \square

We can also provide a lower bound on the sum of a k -carat diamond as the next lemma indicates.

Lemma B.1. *If the diamond of size $s_1 \times s_2 \times \dots \times s_d$ has k -carats, then its sum is at least $k \max(s_i) \text{ } i \in \{1, 2, \dots, d\}$.*

Proof. We know that the sum is at least as large as the sum of $\max(s_i)$, and each s_i is at least as large as k , from which the result follows. \square